A Simplified Interface for Concurrent Processing

Janet D. Hartman
Applied Computer Science Department
Illinois State University
Normal, IL 61704

Abstract *- Teaching the concepts of concurrency is a task requiring an understanding of how the operating system allocates resources for the processes and how concurrent processes may share data. The difficulty of trying to understand the asynchronous execution behavior of processes is compounded by the need to understand the syntax and required parameters for making the system calls. Providing wrappers for system calls that make more intuitive function calls to allocate and manage resources can help students understand the issues involved with shared memory multiprocessing and message passing systems without having to understand the details of the syntax associated with making the system calls. This paper describes a library of C functions, the Parallel Library, that can be used to introduce the concepts of shared memory multiprocessing, message passing, and related synchronization issues, but hide the details of the actual system calls used to implement them. This library has been successfully used to introduce students with minimal operating system knowledge and programming skills to the ideas associated with concurrent processing.*

Introduction

Teaching the concepts of concurrent execution of processes is a task that requires an understanding of how the operating system allocates resources for the processes and how concurrent processes communicate data. The UNIX operating system is well suited for these topics because it supports multiprocessing naturally.

For the novice, the system calls associated with the generation of processes and the allocation and management of interprocess communication structures are difficult to comprehend. Trying to understand asynchronous execution behavior along with mastering the syntax for managing system allocated resources like semaphores and shared memory segments compounds the difficulty.

Providing wrappers for system calls that make more intuitive calls to allocate and manage resources can help students understand the issues involved with shared memory multiprocessing and message passing systems without having to understand the tedious detail associated with the systems calls. What follows is a discussion of a library of C functions, the Parallel Library, that can be used to introduce the concepts of shared memory multiprocessing and message passing, but hide the details of the actual system calls used to implement them. These functions can be used to introduce students to the ideas associated with concurrent processing in many courses, even at the introductory level. The functions are based on the System V UNIX system calls for generating processes and allocating and managing interprocess communication, but could easily be modified for similar POSIX functions.

The Parallel Library

The Parallel Library contains functions for creating and terminating processes along with functions for allocating and managing interprocess communication structures (shared memory and message queues) along with functions for synchronizing access to shared memory (locks and barriers). The inspiration for the Parallel Library came from the textbook by Steven Brawer *An Introduction to Parallel Programming.* The functions and the interfaces are described below. The functions are designed to be used in an application where one process generates one or more child processes. As presented, they will not work with processes that are generated by separate parent processes and are totally unrelated. The complete code, a user's manual, and sample programs using the functions can be obtained from the Web site *http:// cs.acs.ilstu.edu/~hartman/sunprogs/parallel/*

Creating and Terminating Processes

In UNIX the *fork()* system call is used to create a clone of the calling process. UNIX *fork()* always returns 0 to the child process that it creates and the UNIX assigned child process identifier to the calling process. Multiple calls to *fork()* generate a series of child process identifiers that are not necessarily sequential.

The function call for the creation of processes in the Parallel Library is based on the UNIX system call *fork()*. This function, ***int fork_procs(int nprocs)***, produces nprocs - 1 clones of itself by executing the UNIX fork() system call nprocs-1 times. The function returns a logical process identifier for each of child processes so that they are numbered 1,2,3,...nprocs - 1

0-7803-5643-8/99/$10.00 © 1999 IEEE

and a logical identifier of 0 to the caller. The logical numbering of the children and the parent process is much more conducive to assigning work to the processes in a logical, sequential manner based on a logical process identifier. In the example shown below, the five child processes will report values in the range 1..4 (not necessarily in that order) and the calling process will report a value of zero:

```
int proc_id;
int nprocs = 5;

proc_id = fork_procs(nprocs);
printf("My logical id is %d\n",proc_id);
```

Figure 1

A process generally terminates naturally by completing the code that it is executing. Since all processes generated in an application are asynchronously executing and independent of each other, it is possible that the parent process will terminate before the child processes it creates. To prevent this, the UNIX system call *wait()* can be called for each child process generated to force the parent process not to terminate before its children. In the Parallel Library, the function ***void join_procs(int nproc, int proc_ id);*** uses the UNIX system call *wait()* to prevent the parent process from terminating before its children. After this function completes, only the original parent process has not yet terminated. The function should be called when the remainder of the work to be done in an application, like output, needs to be completed by only one process.

Interprocess Communication

Since processes are independent entities with separate address spaces, there is no inherent way for them to share data. Any sharing of data must be done explicitly by creating one or more interprocess communication structures that each has permission to read and modify. Three interprocess communication structures are discussed in this paper: shared memory segments, semaphores, and message queues. Each of these must be allocated by the operating system and requires a system call with a fairly complex interface like the one for performing semaphore operations shown in Figure 2 below. It is assumed in all the functions in the Parallel Library that the processes that are communicating share an ancestor.

```
int semop(int semid, struct sembuf semoparray[],size_t ops);
```

Figure 2

Allocating Shared Memory

One way to share data is to request a shared memory segment from the operating system. Only one process needs to request allocation of a memory segment, but many can attach that segment to their data space. In UNIX two system calls, *shmget()* and *shmat()*, are necessary to allocate a shared memory segment and associate it with data in a process. The Parallel Library function ***void * shared(int num_bytes, int * shmid);*** combines these into one operation. It dynamically allocates a block of shared memory of size *num_bytes* bytes, modifies the *shmid* parameter to be the unique identifier that is assigned by UNIX to the shared memory segment, and returns a pointer to the shared memory allocated. The value assigned to the parameter *shmid* is the shared memory segment identifier that is needed to de-allocate the segment when it is no longer needed. The pointer that is returned must be assigned to a variable in the program that is to be stored in shared memory. Note that the result type is a void pointer that must be typecast to the type of variable to be stored there. Note also that only the data is shared; the pointer variable that contains the address of the memory for the data is local to the process that requests the shared memory segment. Thus, each process that attaches to the memory segment has its own reference to it. The function *shared* needs to be called only once for each segment of shared memory allocated and should be called before any child processes are created. Children forked after this call will inherit a copy of the identifier of the shared memory segment that was assigned by the operating system.

An example of allocating an integer array of ten elements which is referenced by * *myarray* is shown in figure 3 below followed by a segment of code to assign values to the elements in the array The *sizeof* function is used to determine the number of bytes needed to store an element of the specified type.

```
int * my_array;
int   shmid;

my_array = (int *) shared(10 * sizeof(int),&shmid);
for (i = 0; i < 10; i++)
        myarray[i] = i;
```

Figure 3

Synchronizing Access to Shared Memory

Since shared memory variables are designed to be accessible to multiple processes, it may be necessary to make sure that the values assigned to them cannot be manipulated by multiple processes simultaneously, particularly updates to the variable. While concurrent reading of shared variables is generally not an issue, the interleaving of reads and writes to a variable by

multiple processes where order matters is a problem. In both cases, the accesses to the shared variable must be synchronized. This requires a structure with atomic operations that each process tests before entering the section of code where the shared memory variable is manipulated (read or written). The segment of code, referred to as a *critical section*, is executed exclusively be each process sequentially, but not necessarily in order. No two processes can be executing the code concurrently.

The atomic structure, referred to as a *lock*, is created using an interprocess communication structure called a *semaphore*. If a process is in a critical section, any other processes that test the lock must wait until the lock has been unlocked to enter the critical section. A lock is a binary counter, having a value of 0 or 1. To lock it, it must be decremented; to unlock it, it must be incremented. Excessive use of locks can slow program execution considerably. There are three functions that manipulate a lock: one to acquire and initialize it, one to lock it, and one to unlock it.

The function ***int get_lock();*** allocates a binary semaphore for a lock and initializes it to be unlocked. It returns the unique UNIX assigned value for the semaphore. This function only needs to be called once for each lock and should be called before any child processes have been created.

The function ***void lock(int lok);*** decrements the binary semaphore that is used to implement the lock identified by the variable *lok*. When a process calls this function, subsequent processes that call it before the original calling process releases it are suspended until the original process unlocks the lock. A precondition for the function is that the lock has been previously unlocked by a call to the function *unlock*.

The function ***void unlock(int lok);*** increments the binary semaphore that is used to implement the lock identified by the variable *lok*. When a process calls this function, the semaphore is released (incremented) allowing another process to lock the lock and enter the critical section of code. A precondition for this function is that the lock has been locked.

Sometimes it may be necessary to halt a group of processes executing a segment of code until all have reached a specified point in order to prevent one or more from racing ahead and computing erroneous results. The structure used to do this is referred to as a *barrier* and has been implemented as part of parallel languages for parallel computers like the Sequent. The basis for the barrier is a counting semaphore which keeps track of the number of processes that have called the barrier function and are currently waiting to exit the function. Once all have called the function, they are released to continue execution of the remainder of the program. There are two functions in the Parallel Library to allocate and manipulate barriers.

The function ***void init_barrier(int bar[], int blocking_number);*** allocates and initializes a barrier array to synchronize the execution of *blocking_number* processes. The value *blocking_number* is stored in one element of the barrier array *bar* for later use by the function *set_barrier*. This function only needs to be called once when a barrier is created. It should be called before any child processes are created.

The function ***void set_barrier_(int bar[]);*** sets the barrier represented by the array *bar* and determines when *blocking_number* processes have called it. When all have "arrived" (called the function), all are allowed to proceed.

Allocating a Message Queue

Another way to share data between processes is to use a message queue. A message queue is a shared buffer into which processes can deposit data (messages) or from which they can remove data. Message queues are also allocated by the operating system. The Parallel Library function ***int new_message();*** allocates a message queue and returns the unique UNIX assigned identifier for the queue. Messages and message queues have defined lengths in bytes which can be determined by typing *sysdef* at the command line prompt.

Sending and Receiving Messages

Data is communicated between processes by sending and receiving messages via the message queue. Each message sent must have an identifier field so that a requesting process can receive specifically identified messages rather than arbitrary ones. There may be many messages in a queue, each of which has a different identifier field.

Data access using message queues is not as difficult to synchronize as it is with shared memory. If a process wishes to acquire data and it has not yet arrived in the message queue, the requesting process suspends until the data arrives in the queue. If the data never arrives in the queue then the process will never be able to complete execution. This situation is called *deadlock*.

The function ***void send(int msgid, struct msgformat * msg, int bytes);*** sends the message stored in the buffer *msg* to the queue identified by *msgid* and returns the number of bytes actually sent. The structure *msgformat* represents the message to be sent and can be defined in any way that the user wishes as long as the first field is an integer that identifies the tag identifier for the message. An example of a message buffer is shown in figure 4 below. The parameter *bytes* indicates how many bytes are being placed in the queue, usually the size of the message buffer.

```
struct msgformat {
                    int message_id;
                    int count;
                    char myname[32];
                    }
```

Figure 4

The function ***int receive(int msgid, struct msgformat * message, int bytes, int type);*** receives a message from the message queue *msgid* into a message structure *message* that contains *bytes* amount of storage and returns the actual number of bytes received. The parameter *type* is used to identify a message to determine if a message of that type is currently in the queue, since the queue can store many types of messages. In searching for a message to remove from the queue, this function applies the rules below to the value in the parameter *type*:

- When *type* is 0, it removes the first message from the queue.
- When *type* is greater than 0, it removes the first message that matches *type* from the queue.
- When *type* is less than 0, it removes the first message with a type that is lower than the absolute value of *type* requested.

Removing System Allocated Resources

When message queues, semaphores and shared memory segments are allocated using the functions in the Parallel Library, these resources are allocated by the kernel on behalf of the calling program. They have unique identifiers and have owners and permission structures similar to other structures in UNIX. These resources are allocated outside the data space of any process that has access to them and, therefore, ***not*** reclaimed automatically by the operating system when the process that allocated them terminates; they are kernel persistent. In addition, each computer has a limited supply of them. The command *sysdef* can be executed at the command line prompt to determine what these limits are on a particular system.

In order to determine what resources have been allocated to users, the command *ipcs -smq* can be executed at the command line prompt. This generates a list of all system resources that have been allocated indicating the unique system identifier of each and its type.

When a user exits the system, it is possible that resources are still allocated to (owned by) the user and, thus, not accessible to other users. The reclamation of system allocated resources must be performed explicitly. Ideally, this should be performed by a process that has access to the resources before it terminates. Functions to remove shared memory segments, semaphores, and message queues within a program are included in the Parallel Library. Each of the functions, **void remove_shm(int shmid);, void remove_sem(int semid);,** and **void deleteq(int msgid);** removes exactly one resource; a call must be made for each system resource of that type that has been allocated in a program. The argument to the function in each case is the unique identifier assigned to the resource by the operating system when it was allocated.

Even if a programmer is careful and adds function calls to remove system allocated resources, a program may abnormally terminate. In this case the resources still exist and some other method must be used. Each resource could be removed manually using the command *ipcrm* at the command line prompt. To use *ipcrm*, the user must know the unique identifier assigned to the process when it was allocated which can be determined by executing *ipcs –smq* at the command line prompt, as mentioned earlier.

Because the removal of a system allocated resource using *ipcrm* is tedious and must be performed independently for each resource allocated, the author wrote the following C-shell script which removes any system allocated resources that the user has been assigned. It is a good idea to add execution of this script to the *.logout* file so that a user's system resources will be cleaned up when he exits the system.

The Cleanup Script

```
#!/bin/csh
set outvar = `ipcs -mb |grep $LOGNAME|awk '{print $2}'`
foreach ipc_mem ($outvar)
ipcrm -m  $ipc_mem
echo $ipc_mem
end
echo MEMORY SEGMENTS REMOVED
set outvar = `ipcs -sb |grep $LOGNAME|awk '{print $2}'`
foreach ipc_sem ($outvar)
ipcrm -s $ipc_sem
echo $ipc_sem
end
echo SEMAPHORES REMOVED
set outvar = `ipcs -sq |grep $LOGNAME|awk '{print $2}'`
foreach ipc_qem ($outvar)
ipcrm -q $ipc_qem
echo $ipc_qem
end
echo MESSAGE QUEUES REMOVED
echo CLEAN UP IS DONE
```

Sample Assignments

The Parallel Library can be used to teach students how concurrent processes behave and communicate. They can also be used as a simulator for a shared memory multiprocessor where each process generated can be considered as executing on a single processor. In this manner, students can learn about parallel programming in a shared memory environment. Two assignments are described below. The first was assigned in a sophomore level class in which introductory concepts in computer

organization, operating systems and data communications are presented. The second was assigned in a junior level computer organization course.

Assignment Example One

Your task is to write a C program in which a segment of text is examined and the total number of vowels in the text is calculated. The list of vowels is: *a, e, i, o, u*. Your program should recognize both lower and upper case vowels. The problem should use 5 processes to recognize the vowels and count the number of each in the text. Data is shared by the processes by accessing a shared memory segment.

The main program should generate 5 child processes (6 including itself). The main process should

- read in the segment of text to be examined and print it out
- allocate a shared memory segment for the total number of vowels in the segment of text
- allocate a lock variables to protect the critical segment of code when each of the child processes is updating the total vowel count
- wait on the child processes to be terminated before doing the next step
- print out the total number of vowels found in the string

Each child process should take responsibility for counting the number of times one of the vowels occurs in the string. Each process should step through the string and keep count in a local counter (it has not been put in shared memory). All of this takes probably less than 50 lines of code. A sample pseudocode solution is attached (*not in this paper*).

Assignment Example Two

Your task is to write a program that will evaluate a 20 term polynomial in x, where x = 2. You should assign processes to different terms in the polynomial in some systematic way to calculate the value of x to the power of the term times its coefficient ($4x^2 = 4 * x * x$). For example, if there are 4 CPU's available, you could give the first process the first 10 terms, the second process the second ten terms, and the third CPU the third ten terms. You could also divide the work up in a way that the first process gets the 1, 5, 9, 13,… terms, the second process gets the 2,6,10,14,…terms, etc. You could also consider the terms as a pool of potential tasks and let a process perform one and then go get another from the pool. When the pool is empty all tasks are finished. Each process should maintain a local sum value in which it keeps the sum of the terms which it has evaluated. Thus, at the end of the processing, there should be 4 local sum values if you used 4 processes. Each process should contribute its local sum to the sum in a critical section. An algorithm for polynomial evaluation is presented on the attached sheet (*not in this paper*).

Summary

A set of C functions that serve as wrappers to UNIX system calls can be used to increase students' understanding of how concurrency works in an operating system. This paper has described a set of wrapper functions for manipulating concurrent processes using shared memory and message queues. A similar package could be developed to teach other complex topics like socket programming. Such tools make complex ideas accessible even to younger students.

References

Brawer S., *Introduction to Parallel Programming*, 1989.
Stevens, W.R., *Advanced Programming in the UNIX Environment*, 1993.